HyperFlow: A Processor Architecture for Timing-Safe Information-Flow Security

Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, G. Edward Suh
Cornell University
Ithaca, NY 14850, USA
af433@cornell.edu, yz424@cornell.edu, andru@cs.cornell.edu, suh@ece.cornell.edu

***Abstract*—This paper presents HyperFlow, a processor that enforces secure information flow, including control over timing channels. The design and implementation of HyperFlow offer security assurance because it is implemented using a security-typed hardware description language that enforces secure information flow. Unlike prior information-flow secured processors that aim to strictly enforce noninterference, HyperFlow supports complex information flow policies that can be configured at run time, and provides support for secure interprocess communication (IPC) and system calls. The architecture also offers a new model for process isolation in which memory protection is provided via information flow control with strong security assurance while allowing IPC and shared memory. HyperFlow is designed to support practical applications and system architectures. It therefore supports decentralized information flow mechanisms that allow controlled communication among mutually distrusting processes, mediated by dynamic, fine-grained labels. Static information-flow verification of such a complex processor architecture poses significant challenges, which require contributions in both the hardware architecture and the security type system. The paper discusses the architecture decisions that make the processor secure and describes a new secure HDL, named ChiselFlow, that allows these decisions to be verified in a lightweight way. The HyperFlow architecture is also prototyped on a fully-featured processor that offers a complete RISC-V instruction set, and is shown to have moderate overhead on area and performance.**

I. Introduction

In modern computer systems, hardware plays a central role in providing isolation for software modules. Protection rings are widely used to isolate supervisor processes from user processes. Recent hardware security architectures such as ARM TrustZone [23] and Intel SGX [8] aim to protect critical software even when the operating system is malicious or compromised. However, the security of these processors relies on the assumption that the underlying hardware properly enforces necessary security properties.

Unfortunately, microprocessors often contain vulnerabilities that allow security-critical software to be compromised by untrusted software. Software-exploitable vulnerabilities in SGX have already been found [24]. Previous studies found vulnerabilities in implementations of Intel VT-d [38] and system management mode (SMM) [37]. Moreover, the recent Spectre [19] and Meltdown [22] vulnerabilities show that even if the hardware is correct in a conventional sense—that it implements a specification—is not sufficient to ensure security. Exploiting subtle timing channels in Intel microprocessors, Meltdown can be used to leak arbitrary kernel data. Therefore, it is important for security to ensure that hardware implementations are free of timing channels.

Prior work has shown that hardware constructed with a security-typed hardware description language (HDL) [20], [43] can provide strong security assurance. Security-typed HDLs can statically prove that the hardware prevents insecure information flows: untrusted signals cannot affect trusted signals, and secret signals cannot affect public ones. Because HDLs produce cycle-level descriptions of the hardware, security-typed HDLs can also prove that the hardware is free of timing channels. Prior work has also shown that security-typed HDLs catch real-world security vulnerabilities [14].

This paper presents HyperFlow, a processor architecture and implementation designed for information-flow security that is verified with a security-typed HDL at design time, providing strong security assurance about its design and implementation. The HyperFlow architecture is carefully designed to remove all disallowed information flows between security levels, including timing channels, and the information flows within the design are statically verified using a security type system. HyperFlow is also implemented as an extension of a fully featured processor with a complete (RISC-V) instruction set.

HyperFlow security policies. Beyond being a practical, realistic processor, HyperFlow also innovates in the security policies it enforces. Unlike prior processors with verified information flow, which only supported simple, fixed 2-point or 4-point policy lattices, HyperFlow can enforce complex application-defined security policies directly in hardware, in line with work on information-flow security in operating systems and programming languages which suggests that real applications need rich lattice policies that can capture complex trust relationships among mutually distrusting principals [5], [12], [26], [41].

We show how to encode complex and dynamic security policies involving both confidentiality and integrity even for applications built from communicating processes serving mutually distrusting principals. By enforcing security policies in hardware, the software component of the trusted computing base is minimized, and strong security assurance is obtained. While practical tagged architectures have previously been built with the ability to encode information flow policies [11], [42], they do not handle timing channels, and their implementations have not been secured with an information flow HDL.

HyperFlow represents rich lattice policies in hardware at the

bit level by introducing *hypercube labels*, in which software-level labels are mapped to points in a hypercube. Hypercube labels enable efficient comparisons between security levels directly in hardware, and they are amenable to static checking in the security-typed HDL.

Controlled downgrading. To be practical, systems based on information-flow security must allow exceptions to noninterference [16]. For example, applications must be able to release the results of computing on secrets. This can be accomplished by *downgrading*, a mechanism for relaxing information-flow policies. Uncontrolled downgrading is dangerous, so the HyperFlow ISA provides instructions for controlled downgrading. Downgrading of confidentiality policies (declassification) is permitted only when it is *robust* [39]—secrets can be released only if the downgrade can be influenced only by the owners of these secrets. Dually, downgrading of integrity policies (endorsement) is permitted only it when it is *transparent* [4]: that is, if the party providing the endorsed data could have read it. Together, these conditions ensure that information flow is *nonmalleable*. Nonmalleable information flow is enforced not only at the ISA level but also at the HDL level, providing similarly strong assurance about the implementation.

Secure interprocess communication. Another novel and challenging feature of HyperFlow is its support for secure communication across trust domains. HyperFlow allows but constrains interprocess communication (IPC) via shared memory. It also supports the secure communication via registers of arguments and return values of system calls. System calls and shared function libraries present another challenge that HyperFlow addresses—both scenarios require a mechanism by which untrusted code can invoke trusted code. HyperFlow introduces an information-flow secure *call gate* [31], [36] mechanism to make cross-domain control transfers secure.

Memory protection. Conventional systems use virtual memory to isolate pages that belong to different applications. However, hardware support for virtual memory is complex and its correctness also depends on other mechanisms such as cache coherence, which is notoriously difficult to implement correctly. HyperFlow replaces conventional memory protection with *security tags* associated with each physical page (or frame) of memory. Security tags are mapped to hypercube labels using a mapping defined by the operating system; accesses to memory are then mediated using hypercube labels. The security of this mechanism is checked in the HDL code at design time. Despite its novel mechanism for memory protection, HyperFlow also provides a virtual-memory interface to support existing operating systems and applications.

ChiselFlow security-typed HDL. To ease the task of information-flow verification, we designed a security-typed HDL called ChiselFlow in which to construct HyperFlow. ChiselFlow is embedded in Scala, and inherits the expressiveness of a complex full-featured language. But ChiselFlow compiles to a small intermediate language that is responsible for the enforcement of security policies, so the trusted component of ChiselFlow is small. Unlike prior secure HDLs, ChiselFlow provides label inference that reduces programmer effort. The hardware designer provides security labels for the inputs and outputs of hardware modules, but labels of internal signals can be omitted. ChiselFlow also supports multiple mechanisms for describing heterogeneously labeled data structures, which are crucial for practical designs. Finally, ChiselFlow separates the confidentiality and integrity components of labels supporting controlled downgrading.

Prototype implementation. We implement HyperFlow as an extension to the RISC-V Rocket processor, and HyperFlow supports a complete RISC-V ISA. Our implementation allows conventional virtual-memory protection to interoperate with HyperFlow's information flow protection. HyperFlow implements performance-critical features that were absent in prior processors secured with an IFC HDL. The prototype implementation shows that the new security features in HyperFlow add moderate area overhead, largely due to the additional storage for security tags, and moderate performance overhead due to timing-channel protection. The HyperFlow implementation is also more fully-featured than prior information-flow secured processors. The timing-safe implementations of these features also type-checks with ChiselFlow, providing strong assurance that the implementation is secure.

II. HDL-level Information Flow Control

Implementing hardware with security-typed hardware description languages (HDLs) is a promising approach to ensuring the hardware is secure. HDL-level information flow control applies techniques from language-based security [29] to hardware design [13], [20], [21], [43]. Variables in the code that describe the hardware design are annotated with security labels, L, which are types describing restrictions on where information contained in that signal can flow. The type system then enforces these restrictions.

Type systems for information flow security can enforce noninterference [16], which ensures that a signal with a label L can only be influenced by signals with labels that are less restrictive than L. For example, if the label `public` is less restrictive than the label `secret`, then a `secret` signal cannot influence a `public` signal. When a label L is no more restrictive than another label L', it is said that L flows to L', written $L \sqsubseteq L'$.

HDLs for information flow security can enforce a particularly strong, timing-safe variation of noninterference [21], [43]. HDLs describe hardware at the register transfer level (RTL) – the code describes new valuations of signals during each clock cycle. Because HDLs give cycle-level descriptions of hardware, the information flow type system can guarantee cycle-level timing-channel freedom.

In this work, we develop a new information-flow-secure HDL, ChiselFlow, and use it to construct a novel and secure processor. ChiselFlow extends Chisel [3], an HDL embedded in Scala. An advantage of the ChiselFlow implementation is that it gains much of the expressiveness of Scala without including this complex language in its trusted computing base. Like Chisel, ChiselFlow generates a simpler, compiled

```
class ExampleIO extends Bundle {
  val id      = Input(UInt(4.W), L(public, trusted))
  val data_in = Input(UInt(32.W), hlvl(id, id))
  val data_out = Output(UInt(32.W), hlvl(id, id))
}

class ExampleModule extends Module {
  val io = IO(new ExampleIO)

  val secretMask = Reg(init = 0x2.U, L(secret, trusted))
  val publicMask = Reg(init = 0x1.U, L(public, trusted))

  when (id confFlowsTo secret) {
    io.data_out := secretMask & io.data_in
  }.otherwise {
    io.data_out := publicMask & io.data_in
  }
}
```

Fig. 1. ChiselFlow example.

intermediate representation that can then be used to produce hardware designs. The intermediate representation (IR) for ChiselFlow is called SIRRTL; it extends Chisel's IR, FIRRTL, with information flow annotations. The enforcement mechanisms of ChiselFlow operate entirely on SIRRTL. In an accompanying technical report [1], we formalize a core subset of SIRRTL and prove that, aside from downgrades, well-typed hardware modules enforce timing-sensitive noninterference. In Appendix A, we discuss the implementation of ChiselFlow and some features that were useful for implementing HyperFlow securely.

Figure 1 shows an example of ChiselFlow code, which looks much like Chisel code aside from the parts in bold font. As in Chisel, ChiselFlow describes hardware modules with classes that extend `Module`. The example shows a module called `ExampleModule`, which takes two inputs: `data_in`, a data value, and `id`, a one-bit flag indicating whether `data_in` is secret or public. The module outputs `data_out`, which is `data_in` masked with a secret value when `id` indicates that `data_out` can observe secrets.

Signals in ChiselFlow are annotated with security labels that have confidentiality and integrity components. Here, the label `L(public,trusted)` means that `id` is fully public and fully trusted. However, ChiselFlow also supports software-defined security policies that depend on the run-time values of variables. The ability to express security policies that change at run time enables hardware implementations with low area overhead, because it allows hardware modules to be shared among security domains over time. For example, `hlvl(id, id)` describes the interpretation of the signal `id` as an information flow label. Although labels of this form can depend on run-time values of signals, the type system relies on a static program analysis to reason about the behavior of these run-time types at design time.

The interface for `ExampleModule` is `ExampleIO`, which describes a record type in which `id`, `data_in`, and `data_out` are records with distinct security labels that have already been described. The register `secretMask` is a fully secret and fully trusted register. The body of ExampleModule is secure because access control ensures that the value of `secretMask` does not flow to `io.data_out` unless `id` flows to `secret`. The program analysis statically determines that under the branch in which this access control is true, the security label of `io.data_out` is `secret`, which permits the assignment from the value that depends on the secret.

III. Security Policies in HyperFlow

HyperFlow enforces information flow security policies directly in hardware. Prior work on label models for information flow security has developed rich policies allow mutually distrusting principals to communicate [5], [12], [26], [41]. These label models represent policies using lattices of information flow labels. HyperFlow can enforce policies described in these models because it can enforce general lattice-based policies.

A. Confidentiality and integrity policies

Information flow labels in HyperFlow support reasoning about both confidentiality and integrity. An information flow label $\ell = (c, i)$ in HyperFlow is a pair of a confidentiality level c and an integrity level i. Confidentiality and integrity levels in HyperFlow both form lattices that are ordered by $\sqsubseteq_C$ and $\sqsubseteq_I$ respectively. The ordering on confidentiality levels specifies constraints on secrecy; if $c \sqsubseteq_C c'$, then c is no more confidential than c'. Similarly, if $i \sqsubseteq_I i'$, then i is at least as trustworthy as i'. The ordering of integrity levels and confidentiality levels is dual: high confidentiality levels are more restrictive than low ones, whereas low integrity levels are more restrictive than high ones. The orderings on confidentiality and integrity levels are lifted to a lattice of labels $\sqsubseteq$; if $c \sqsubseteq_C c'$ and $i \sqsubseteq_I i'$ then $(c, i) \sqsubseteq (c', i')$. We write $C(\ell)$ and $I(\ell)$ to denote just the confidentiality or integrity part of the label respectively.

B. Lattices via bit vectors

In order to support efficient computations and comparisons of labels in hardware, HyperFlow represents lattices over bit vectors. We first explain the ordering of confidentiality levels. Levels are mapped to a point in a hypercube, which is expressed using a bit vector. A bit vector b is split into $d \in D$ dimensions, each of K bits. Bit vectors are then functions from $[1, D]$ to $[0, 2^K - 1]$, and the notation $b(i)$ represents the value in the i^{th} dimension of b. Bit vectors b_1 and b_2 are ordered in the confidentiality lattice, written $b_1 \sqsubseteq_C b_2$ if each dimension of b_1 is numerically less than or equal to the corresponding element of b_2. The join ($\sqcup_C$) and meet ($\sqcap_C$) of two confidentiality components are respectively computed by taking the maximum or the minimum over the corresponding dimensions of each vector. The lattice over bit vectors is defined more formally in Figure 2. As an example, if b_1 and b_2 are each bit vectors of 4, 2-bit dimensions, and b_1 is `10100111` and b_2 is `10010010`, then $b_2 \sqsubseteq_C b_1$. It is straightforward to check that this order has lattice properties (i.e., it is a transitive, reflexive, and antisymmetric partial order). The ordering in the integrity lattice is exactly dual to the ordering in the confidentiality lattice as shown in Figure 3.

We write $(c, i) \sqcup (c', i') \triangleq (c \sqcup_C c', i \sqcup_I i')$ and $(c, i) \sqcap (c', i') \triangleq (c \sqcap_C c', i \sqcap_I i')$ to denote the join and meet over labels respectively. We use $\top$ and $\bot$ to denote a sequence of all `1's` and

$$b \in B = [1, D] \rightarrow [0, 2^K - 1]$$
$$b_1 \sqsubseteq_C b_2 \triangleq \forall d \in [1, D].b_1(d) \leq b_2(d)$$
$$(b_1 \sqcup_C b_2)d \triangleq \max\{b_1(d), b_2(d)\}$$
$$(b_1 \sqcap_C b_2)d \triangleq \min\{b_1(d), b_2(d)\}$$

Fig. 2. Confidentiality ordering over bitvectors.

$$b \in B = [1, D] \rightarrow [0, 2^K - 1]$$
$$b_1 \sqsubseteq_I b_2 \triangleq \forall d \in [1, D].b_1(d) \geq b_2(d)$$
$$(b_1 \sqcup_I b_2)d \triangleq \min\{b_1(d), b_2(d)\}$$
$$(b_1 \sqcap_I b_2)d \triangleq \max\{b_1(d), b_2(d)\}$$

Fig. 3. Integrity ordering over bitvectors.

all `0's` respectively. In the confidentiality order, $\top$ and $\bot$ are completely secret and completely public respectively. In the integrity order, $\top$ and $\bot$ are completely trusted and completely untrusted respectively. The labels $(\bot, \top)$ and $(\top, \bot)$ are the least and most restrictive labels in information flow order ($\sqsubseteq$).

Other representations of lattices in computer systems have been studied [15]. Because HyperFlow uses information flow labels for access control and timing-channel protection, lattice comparisons and computations need to be done throughout the implementation, and the ability to efficiently update and compare labels directly in hardware is particularly important in designing a processor with strong information flow security. Prior representations of lattices such as adjacency lists and matrices are less space-efficient. Other approaches that rely on caching requires software intervention on each lattice operation. The hypercube lattice is most similar to the skeletal representation, also known as the Fidge and Mattern vector clock [18]. However, vector clocks have not been used to represent lattices in hardware in prior work.

C. Nonmalleable downgrading

Systems for information flow control are often intended to enforce noninterference, which prevents all information flows that violate lattice policies. However, noninterference is too restrictive for practical systems. For example, data computed using secrets may eventually need to be released publicly. Noninterference may be weakened through *downgrading* which relaxes information flow labels. Downgrading that weakens confidentiality is said to *declassify* whereas downgrading that weakens integrity is said to *endorse* [40].

Because downgrading weakens noninterference, effort has been made to constrain downgrading to limit its potential to cause harm [30]. In this work, we permit communication that weakens noninterference as long as the downgrading it causes is nonmalleable [4]. Nonmalleable information flow subsumes two security conditions, robust declassification and transparent endorsement. These security conditions have not been enforced by previous hardware mechanisms.

Robust declassification [39] only permits information to be downgraded by parties that have authority over that information. As in prior work on defining robust declassification [4], [6], authority (trust, privilege) is represented by integrity; only a principal at least as trusted as $I(p)$ can declassify data with confidentiality $C(p)$. This constraint is useful for decentralized systems. A principal A can declassify its data to a principal B, and as long as B does not have integrity $I(A)$, B can observe A's data but is prevented from releasing it elsewhere.

In HyperFlow, a process with label ℓ_{cur} can declassify a label ℓ to ℓ' only if the following holds:

$$C(\ell) \sqsubseteq_C C(\ell') \sqcup_C (I(\ell_{cur}) \sqcup_I I(\ell)).$$

This condition follows directly from prior work on defining robust declassification in the context of programming languages [4], [6]. Roughly, it allows the confidentiality $C(\ell)$ of the data being declassified to be "made up for" by the integrity $I(\ell)$ of the data being declassified and the integrity $I(\ell_{cur})$ of the current process. When ℓ can be robustly declassified to ℓ' by a process with label ℓ_{cur}, we write $\ell \xrightarrow[\ell_{cur}]{C} \ell'$.

The dual of robust declassification is transparent endorsement [4]. It sets a maximum confidentiality on endorsements to prevent opaque writes that could enable attacks. A write is opaque if a principal could have written data but not read it. In HyperFlow, a process with label ℓ_{cur} can endorse a label ℓ to ℓ' if,

$$I(\ell) \sqsubseteq_I I(\ell') \sqcup_I (C(\ell_{cur}) \sqcup_C C(\ell)).$$

This condition follows directly from work on defining transparent endorsement for a functional programming language [4]. When ℓ can be transparently endorsed to ℓ' by a process with label ℓ_{cur}, we write $\ell \xrightarrow[\ell_{cur}]{I} \ell'$. When $\ell \xrightarrow[\ell_{cur}]{I} \ell'$ and $\ell \xrightarrow[\ell_{cur}]{C} \ell'$ we say that ℓ can be nonmalleably downgraded to ℓ' by a process with label ℓ_{cur} and we write $\ell \xrightarrow[\ell_{cur}]{} \ell'$ [4].

IV. The HyperFlow Architecture

HyperFlow is realized as a tagged architecture where security labels are explicitly represented as hardware tags for a process, registers, and memory pages. HyperFlow replaces conventional memory protection enforced by virtual memory with security tags that are associated with each physical page (or frame) of memory. Tagged physical memory enables static checking of information flow with a type system. Virtual memory does not ensure noninterference; it is possible for the same physical page to be mapped to virtual addresses owned by distrusting processes. Even if the mapping did ensure noninterference, it would not be possible to prove that noninterference is established purely by inspecting the hardware design, since the mapping is software-defined. The tagged physical memory can also be used to reduce the software trusted computing base by removing the need to rely on virtual memory for process isolation.

The security tags in HyperFlow are information flow labels. By enforcing information flow labels in hardware, HyperFlow can permit isolation among multiple principals that are mutually distrusting, yet communicate. Noninterference precludes communication among mutually distrusting principals, so the information that they are communicating must be downgraded.

However, HyperFlow constrains these downgrades by ensuring that they are nonmalleable [4]. In doing so, HyperFlow ensures that processes cannot leak information that they do not have authority over. Enforcing nonmalleability requires the ability to inspect the integrity and confidentiality of the information being downgraded as well as the principal initiating the downgrading. This is accomplished by making the information flow labels visible in hardware.

A. Process levels

Processes executing in HyperFlow are associated with a level, ℓ_{cur}. The level $C(\ell_{cur})$ represents the greatest level of secrecy that the process can observe, and $I(\ell_{cur})$ represents the most trusted level of information it can affect. In order for the currently executing process to read a page of memory, m, we require $\mathcal{M}_\ell(m) \sqsubseteq \ell_{cur}$, where $\mathcal{M}_\ell$ is a mapping from pages to their information flow labels. Similarly, to write to m, we require that $\ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$.

HyperFlow also associates security labels with registers to facilitate two kinds of communication that are needed in processors: 1) communication between userspace applications and the operating system during system calls, and 2) interprocess communication in memory. During system calls, arguments and return values are communicated between the application and system call handler via registers. HyperFlow permits communication using registers by associating labels with registers and through instructions that downgrade registers' labels. Assuming the application is untrusted, the trusted call handler can endorse the registers storing the arguments after inspecting them. At the end of the system call, the call handler can declassify the registers storing the return values before re-entering the public application.

Because information flow labels are used to enforce security, HyperFlow must ensure that the labels accurately reflect the security of the data they protect. General-purpose register r has security label ℓ_r. Normally, to store the content of r to an address in m, we require $\ell_r \sqsubseteq \mathcal{M}_\ell(m)$. Similarly, loading a word from m into r requires $\mathcal{M}_\ell(m) \sqsubseteq \ell_r$.

Though secure, these invariants sometimes prevent necessary communication among distrusting principals. HyperFlow permits interprocess communication among distrusting principals via shared memory so that it provides a familiar software interface. Writes to and reads from shared memory that would violate noninterference require downgrading. Pages can be downgraded; however, downgrading an entire page is too imprecise for many applications. HyperFlow supports downgrades at the granularity of an individual word with downgrading load and store instructions. These instructions work just like conventional loads and stores except that they downgrade the source data while it is copied. HyperFlow also supports page downgrades for zero-copy sharing of entire pages.

Processes in HyperFlow are also protected against information flow violations caused by the instructions that are fetched by the currently executing process. The active process should not execute low-integrity instructions because this would allow adversaries that the process does not trust to influence the code that the process executes. Similarly, branching conditions that depend on secrets can cause secrets to be released through the instructions that HyperFlow executes. Information leaks through control flow are called *implicit flows*.

HyperFlow prevents implicit flows because ℓ_{cur} also represents the information flow label of the most recently fetched instruction. Branches cannot depend on a register r unless $\ell_r \sqsubseteq \ell_{cur}$. Similarly, for all instructions that write to a register r, HyperFlow requires $\ell_{cur} \sqsubseteq \ell_r$ to ensure that the label of ℓ_r accurately reflects the process that influenced it.

B. Information-flow call gates

The restriction on branch conditions and on writes to registers together prevent an untrusted or secret process from invoking code that is trusted or public. However, untrusted applications need to be able to call trusted code when making system calls, and secret applications need to be able to call public functions from libraries. HyperFlow securely supports control transfers of this form with a mechanism that is analogous to call gates that originated in Multics [31]. Call gates in HyperFlow tightly couple the entry point (program counter) that initiates the code with an information flow label that represents the privilege level of that code. A process at level ℓ_{cur} can register a call gate at level ℓ' as long as $\ell_{cur} \sqsubseteq \ell'$. Another process can then invoke a call gate, at which point the program counter is set to the entry point of the gate and ℓ_{cur} is set to the level at which the gate was registered. To allow protected returns from call gates, invoking a call gate also pushes the previous program counter value and level of ℓ_{cur} onto a hardware stack. The executing process can then invoke a return instruction to pop the stack, jumping to the old pc value and privilege level.

Call gates in HyperFlow are unique in that conventional hierarchical privilege levels are replaced with more general lattice-model information flow labels. By generalizing privilege levels, HyperFlow securely supports control transfers with fewer privilege changes than in a conventional processor while simultaneously providing more fine-grained separation of privilege. For example, in a system managed by a microkernel running on HyperFlow, a network driver can register a call gate at a security level ℓ_{net} that is incomparable with other components of the microkernel. When an application wishes to send a packet over the network, it can directly invoke the call gate transferring immediately to ℓ_{net}. In a conventional processor, the network driver can either run in supervisor mode, in which case the application must implicitly trust the entire kernel, or the network driver can run in userspace. In the second case, the application must first make a system call causing a transition to supervisor mode before the kernel delegates to the userspace driver. In this case, the application must both trust the kernel to delegate to the driver, and there is a performance penalty because of the extra privilege changes.

Using just a single level, ℓ_{cur}, for a given process is often sufficient. However, other applications require the ability to operate on data within a space of information flow labels.

To permit flexibility with the label of executed instructions, HyperFlow allows the active process to move the level of ℓ_{cur} within a space of labels bounded by ℓ_{lwr} and ℓ_{upr}. When setting the value of ℓ_{cur}, we require $\ell_{lwr} \sqsubseteq \ell_{cur} \sqsubseteq \ell_{upr}$. For a process executing with a space of labels, $C(\ell_{upr})$ and $I(\ell_{upr})$ represent the most secret and most trusted information that the process can observe and affect respectively. On the other hand, $C(\ell_{lwr})$ and $I(\ell_{lwr})$ represent the least secrecy the process can claim it has observed and the least trustworthy information that it can be influenced by.

C. Instruction set extensions

HyperFlow introduces new instructions as well as new control and status registers. Security levels in HyperFlow are represented as a pair of confidentiality and integrity components as described in Section III. Levels ℓ_{lwr}, ℓ_{cur}, and, ℓ_{upr} are each stored in control and status registers (CSRs) and are accessed with conventional CSR instructions. The registers that store ℓ_{lwr} and ℓ_{upr} define the bounds for a process. To prevent a process from circumventing its own bounds, the bounds can only be modified when the processor is in the most public and trusted level, that is $\ell_{cur} = (\bot, \top)$. However, ℓ_{cur} can be modified at any level as long as $\ell_{lwr} \sqsubseteq \ell_{cur} \sqsubseteq \ell_{upr}$.

The new instructions are summarized in Table I. The first column describes the instruction name and operands, the second column describes restrictions that must be satisfied when executing instructions, and the third column describes what the instruction does if the restrictions are satisfied. We overload the notation r to denote the value stored in register r. As before, ℓ_r denotes the label associated with r and $\mathcal{M}_\ell(a)$ denotes the label of the page containing address a.

The instructions DECLREG and ENDOREG downgrade registers. The DECLREG instruction declassifies the value stored in r_1 to the confidentiality level stored in r_2, but it permits the declassification only if it is robust. The first restriction prevents implicit flows by ensuring that ℓ_{cur} can write to the new level of r_1. The second restriction ensures that r_1 can be robustly declassified from ℓ_{r_1} to $(r_2, I(\ell_{r_1}))$.

The last restriction is more subtle – it prevents potential information flow violations that might be caused by the use of r_2 as an argument. The register labels and memory labels are fully public and fully trusted. In most work on secure information flow, labels are public and trusted; otherwise, merely inspecting the labels releases information, and if the labels are not trusted, it is hopeless to rely on them for security. Because this instruction allows the value stored in r_2 to influence a label, it must be permitted to influence fully public and trusted data. A natural way to ensure this is to simply require that $\ell_{r_2} = (\bot, \top)$. However, this restriction would often require extra instructions to first downgrade r_2 before downgrading r_1. Instead, we enforce a less restrictive, but equally secure condition—it must be possible to downgrade r_2 to $(\bot, \top)$ using robust declassifications and transparent endorsements. This relaxed restriction does not weaken security because when the restriction on the label of r_2 holds, it is always possible to first downgrade the label of r_2.

The instruction RSTREG allows a process to reclaim a register without downgrading by setting the level of the register r_1 to ℓ_{cur}. In order to avoid possibly downgrading the value stored in r_1, r_1 is cleared. Because this instruction takes no arguments other than r_1, and it happens unconditionally, no restrictions on this instruction are necessary. This instruction is useful for easily resetting the labels of registers because it does not impose any restrictions.

The LWDWN instruction works like a normal load word instruction, but it relaxes the restrictions on information flow labels. It permits the load if the value of the page from which the data is loaded could be downgraded to the label of the destination register, and to ℓ_{cur}. Similarly, SWDWN works like a store instruction that permits the store if the register contents could be downgraded to the label of the destination page. Both instructions are useful for interprocess communication via shared memory.

The memory levels can be reset by totally trusted and public software through a SETMEM instruction, which takes two arguments: the page-aligned physical address in register r_1 and confidentiality and integrity components in r_2. SETMEM sets $\mathcal{M}_\ell(m)$ to r, where r is the value stored in r. The SETMEM instruction can only be executed when $\ell_{cur} = (\bot, \top)$. Trustworthy software that uses this instruction should clear the contents of the page to prevent implicit downgrades.

Entire pages can also be declassified/endorsed by user-space applications through the DECLMEM and ENDOMEM instructions, which are similar to SETMEM except that they require the changes in memory levels to be robust/transparent as in DECLREG and ENDOREG. As with DECLREG and ENDOREG, information flow violations through labels are also prevented by requiring that the arguments that influence labels can be downgraded securely.

The REGLGATE instruction registers a new call gate with a pc value of r_1 and a label of r_2 by adding it to a table T, that stores call gates by mapping pc values to labels. The first restriction, $(\ell_{cur} \sqcup \ell_{r_1} \sqcup \ell_{r_2}) \sqsubseteq r_2$, checks that the process creating the gate and the arguments from which the gate is constructed are no more secret and are at least as trusted as the label of the gate. The entries in the call gate table are public and trusted (though the labels of individual gates may be more restrictive), because processes that attempt to use call gates must be able to see whether or not they exist. Therefore, the last two restrictions check that the active process can downgrade the register arguments to public and trusted because they influence the creation of a call gate entry.

The LCALL and LCALLR instructions execute a call gate and have the same instruction formats as conventional JAL and JALR instructions. The LCALL instruction specifies the call-gate entry point with an immediate that is added to the current pc value, whereas the LCALLR instruction specifies the entry point by adding an immediate to a register argument. For both instructions, if the specified entry point is found in the call gate table, the address of the instruction following the call and the value of ℓ_{cur} prior to the call are pushed to a hardware stack S for return addresses and labels. The processor then sets the

Instruction	Restrictions	Behavior
`DECLREG R1, R2`	$\ell_{cur} \sqsubseteq (r_2, I(\ell_{r_1}))$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{C} (r_2, I(\ell_{r_1}))$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$C(\ell_{r_1}) \leftarrow r_2$
`ENDOREG R1, R2`	$\ell_{cur} \sqsubseteq (C(\ell_{r_1}), r_2)$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{I} (C(\ell_{r_1}), r_2)$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$I(\ell_{r_1}) \leftarrow r_2$
`RSTLREG R1`	None.	$\ell_{r_1} \leftarrow \ell_{cur}$ $r_1 \leftarrow 0$
`LWDWN R2, IMM(R1)`	$\mathcal{M}_\ell(r_1 + \texttt{IMM}) \xrightarrow[\ell_{cur}]{} \ell_{r_2}$ $\mathcal{M}_\ell(r_1 + \texttt{IMM}) \xrightarrow[\ell_{cur}]{} \ell_{cur}$	$r_2 \leftarrow \mathcal{M}(r_1 + \texttt{IMM})$
`SWDWN R2, IMM(R1)`	$\ell_{r_2 \sqcup \ell_{cur}} \xrightarrow[\ell_{cur}]{} \mathcal{M}_\ell(r_1 + \texttt{IMM})$	$\mathcal{M}(r_1 + \texttt{IMM}) \leftarrow r_2$
`SETMEM R2, IMM(R1)`	$\ell_{cur} = (\bot, \top)$	$\mathcal{M}_\ell(r_1) \leftarrow r_2$ $\mathcal{M}(r_1) \leftarrow 0$
`DECLMEM R2, IMM(R1)`	$\ell_{cur} \sqsubseteq (r_2, I(\mathcal{M}_\ell(r_1 + \texttt{IMM})))$ $\mathcal{M}_\ell(r_1 + \texttt{IMM}) \xrightarrow[\ell_{cur}]{C} (r_2, I(\mathcal{M}_\ell(r_1 + \texttt{IMM})))$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$C(\mathcal{M}_\ell(r_1)) \leftarrow r_2$
`ENDOMEM R2, IMM(R1)`	$\ell_{cur} \sqsubseteq (C(\mathcal{M}_\ell(r_1)), r_2)$ $\mathcal{M}_\ell(r_1 + \texttt{IMM}) \xrightarrow[\ell_{cur}]{I} (C(\mathcal{M}_\ell(r_1 + \texttt{IMM})), r_2)$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$I(\mathcal{M}_\ell(r_1)) \leftarrow r_2$
`REGLGATE R1, R2`	$(\ell_{cur} \sqcup \ell_{r_1} \sqcup \ell_{r_2}) \sqsubseteq r_2$ $\ell_{r_2} \xrightarrow[\ell_{cur}]{} (\bot, \top)$ $\ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$T[r_1] \leftarrow r_2$
`LCALL IMM`	None.	$S \leftarrow S :: (\texttt{pc} + 4, \ell_{cur}, \ell_{lwr}, \ell_{upr})$ $\texttt{pc} \leftarrow \texttt{pc} + \texttt{IMM}$ $\ell_{cur} \leftarrow T[\texttt{pc} + \texttt{IMM}]$
`LCALL IMM(R1)`	$\ell_{r_1} \xrightarrow[\ell_{cur}]{} (\bot, \top)$	$S \leftarrow S :: (\texttt{pc} + 4, \ell_{cur}, \ell_{lwr}, \ell_{upr})$ $\texttt{pc} \leftarrow r_1 + \texttt{IMM}$ $\ell_{cur} \leftarrow T[r_1 + \texttt{IMM}]$
`LRET`	None.	$(pc, \ell_{cur}, \ell_{lwr}, \ell_{upr}) \leftarrow \texttt{tail(S)}$ $S \leftarrow \texttt{head(S)}$
`SETBOUNDS`	$\ell_{cur} = (\bot, \top)$	$\ell_{cur} \leftarrow \ell_{ncur}$ $\ell_{lwr} \leftarrow \ell_{nlwr}$ $\ell_{upr} \leftarrow \ell_{nupr}$

TABLE I
NEW INSTRUCTIONS ADDED IN HYPERFLOW.

pc value to the entry point of the gate and sets ℓ_{cur} to the label of the gate. If the gate does not exist, the instruction is converted to a NOP. The instruction LRET pops the stack S and returns to the most recent pc value and label.

Finally, the SETBOUNDS instruction permits software that is fully public and fully trusted to set the label bounds. There are CSRs called ℓ_{ncur}, ℓ_{nlwr}, ℓ_{nupr} that privileged software can read and write normally. The SETBOUNDS instruction atomically copies these CSRs to ℓ_{cur}, ℓ_{lwr}, and ℓ_{upr} respectively in a single cycle. This instruction is necessary, because writing to an individual bound register might otherwise temporarily violate the invariant, $\ell_{lwr} \sqsubseteq \ell_{cur} \sqsubseteq \ell_{upr}$.

D. Semantic changes to existing instructions

In addition to the new instructions, HyperFlow also changes the semantics of existing instructions in order to ensure that the policies described by the information flow labels are enforced. To enforce these polices, a set of invariants must hold for each instruction that is executed. The invariants depend on the kind of instruction being executed. For example, different invariants hold for arithmetic instructions and memory instructions. Table II summarizes these invariants. The invariants serve two purposes: 1) to implement memory protection, and 2) to ensure that the labels of the registers and memory pages accurately capture the secrecy and integrity of the data they protect.

Instruction Type	Invariant
Load instructions	$\ell_{r_a} \sqcup \mathcal{M}_\ell(m) \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$ $\wedge \mathcal{M}_\ell(m) \sqsubseteq \ell_{cur}$
Store instructions	$\ell_{r_a} \sqcup \ell_{r_v} \sqcup \ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$
Execute unit	$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$
Value-dependent branches	$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqsubseteq \ell_{cur}$
All instructions	$\mathcal{M}_\ell(m_i) \sqsubseteq \ell_{cur}$

TABLE II
INSTRUCTION INVARIANTS ENFORCED BY HYPERFLOW.

Memory protection is enforced by ensuring that when a process with label ℓ_{cur} loads from a page m, $\mathcal{M}_\ell(m) \sqsubseteq \ell_{cur}$,

to prevent reads that would violate security. This is explicitly enforced on loads. On stores to m, we require $\ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$ which is subsumed by the invariant enforced by store instructions listed in the table.

For all instructions regardless of type, HyperFlow must enforce the condition $\mathcal{M}_\ell(m_i) \sqsubseteq \ell_{cur}$, where m_i is the memory page where the instruction is fetched from. This prevents information from leaking to the process via the fetched instruction.

The rest of the invariants preserve the accuracy of the information flow labels. For load instructions, condition

$$\ell_{r_a} \sqcup \mathcal{M}_\ell(m) \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$$

must also hold, where r_a is the source register that contains the base address and m is the page which contains the data being loaded. This invariant ensures that the level of the destination register accurately reflects the level of the data it stores. Similarly, store instructions require

$$\ell_{r_a} \sqcup \ell_{r_v} \sqcup \ell_{cur} \sqsubseteq \mathcal{M}_\ell(m)$$

where r_v is the register that contains the value being written, and m is the page being written to. This invariant ensures that the policy described by the level of the page being written to is also not violated by the data being written to the page or by the address.

Computation instructions such as arithmetic and logical instructions, multiplication and division, and floating-point instructions, perform a computation on arguments and write the result back into a destination register. For these instructions,

$$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqcup \ell_{cur} \sqsubseteq \ell_{r_d}$$

must hold, where r_{s1} and r_{s2} are the source registers and r_d is the destination register. The data is influenced by the values of both the source registers (which are bounded by $\ell_{r_{s1}}$ and $\ell_{r_{s2}}$) as well as the process causing the instruction to execute (which is bounded by ℓ_{cur}).

Value-dependent control-flow instructions such as conditional branches must enforce the invariant

$$\ell_{r_{s1}} \sqcup \ell_{r_{s2}} \sqsubseteq \ell_{cur}.$$

Because ℓ_{cur} represents the security level of the current control flow (program counter) as well as the security level of a process, a change to the program counter can only be affected by information that can flow into ℓ_{cur}. This invariant prevents branch instructions that would violate the information flow policy.

V. HyperFlow Microarchitecture and Labeling

This section describes the microarchitecture extensions necessary for HyperFlow and how they are be labeled in the secure HDL. The HyperFlow instruction set architecture can be realized by many implementations and microarchitectures. Here, we discuss our prototype implementation based on the RISC-V Rocket Chip processor. The HyperFlow implementation includes many advanced microarchitecture features such as a pending store buffer, pipelined caches, branch prediction, virtual memory, and atomic memory operations, which were not studied in previous information-flow-secured processor designs. Appendix C provides more details on the design tradeoffs we considered to make the HyperFlow implementation pass the information flow security analysis performed by the type system, and how the process of implementing hardware with a security-typed HDL differs from that of conventional hardware designs.

A. Labels in the core and label bypassing

In the processing core, the security label of the current process (ℓ_{cur}) is stored in a new control status resister (CSR). The confidentiality and integrity components, $C(\ell_{r_i})$ and $I(\ell_{r_i})$, of general purpose registers r_i are stored in register banks adjacent to the registers. These label registers can be modified only by the `DECLREG` and `ENDOREG` instructions, which are guarded by logic that checks the non-malleability conditions, and the `RSTLREG` instruction which can only set the label of a register to ℓ_{cur}.

The HyperFlow core supports data bypassing. To function correctly, the security labels must be bypassed with the data. For immediate values, the bypassed label is ℓ_{cur}. For a value from a register or a cache, its label travels with the bypassed data. The bypassed labels are themselves labeled with ℓ_{cur} because they might be stalled or updated by the current process.

B. Memory and cache labels

The memory page labels ($\mathcal{M}_\ell(m)$) are stored in an on-chip table that maps page numbers to labels. Initially, every page is mapped to the most public and trusted label $C(\bot) \wedge I(\top)$. The `SETMEM`, `DECLMEM`, and `ENDOMEM` instructions issue memory transactions that modify the memory label table. The current security label (ℓ_{cur}) is attached to each memory transaction so that information flow and downgrading can be checked in the memory system. For a system with large off-chip memory, the memory page labels may be stored in off-chip memory along with data.

In the data cache, a data label is added to each cache line to track the memory label for the physical address stored in the cache line. The memory label is appended to a cache refill transaction from memory. The data cache is blocking, so memory tags are always brought into the cache before any data is modified or returned to the core. For a load, the cache only returns data if the data label of the accessed cache line flows to ℓ_{cur}. The core updates the destination register only if the label of the returned cache data flows to the label of the destination register. The security of a store is enforced by checking that the label of a pending store buffer entry flows to the label of the cache line, and that the label of a memory transaction flows to the memory page label.

The `LWDWN` instruction may be used to load a value with downgrading and performs three non-malleability checks. The cache checks that the value can be downgraded to ℓ_{cur}. Then, the core checks that the data label of the cache response can

be downgraded to the destination register's label; two checks are added, one near the bypassed data and the other near the register file writeback.

C. Timing-channel protection

ℓ_{cur} is also used as a timing label to prevent timing channels through microarchitectural state. That is, $C(\ell_{cur})$ is an upper bound on the level of secrecy that the process is permitted to observe by measuring timing. Any microarchitectural state that influences the timing of instructions is protected by ℓ_{cur}. Cache entries, in-flight instructions and cache transactions, translation-lookaside buffer (TLB) and page table walker (PTW) entries, and branch predictor state are examples of state that influences instruction timing. Because the security type system in ChiselFlow enforces timing-sensitive non-interference, timing channels must be removed for the hardware to type-check.

When the value of ℓ_{cur} moves downwards in the lattice, the level of secrecy that the process can observe is decreasing. HyperFlow must prevent secrets owned by the previous level of ℓ_{cur} from leaking to the new one. The processor pipeline is drained to prevent high instructions from stalling low instructions as well as other subtle timing channels through register bypassing. In-flight transactions in pipelined caches are also drained when ℓ_{cur} is lowered.

The pending store buffer in the data cache also introduced a subtle timing channel that we did not initially expect. Outstanding cache-write requests in the pending store buffer are serviced opportunistically when there is no in-flight read request. The store buffer can cause a stall either when the content of the buffer might have a read-after-write hazard or when the buffer is full. To prevent a timing channel, we enforce that all entries of the pending store buffer must have the same label, and the buffer is drained before lowering ℓ_{cur}.

Caches may also cause timing channels when they are shared among security levels. For instruction caches, the timing channel can be removed by simply clearing and invalidating cache lines when lowering ℓ_{cur}. However, in the data cache, dirty cache lines must be written back when they are evicted, and cannot be simply invalidated. In our implementation, we require software to issue a cache flush instruction to write-back dirty cache blocks before executing an instruction to lower ℓ_{cur}. When ℓ_{cur} is lowered, the data cache is invalidated in a single clock cycle.

While our prototype implementation uses flushing to remove cache timing channels, cache partitioning can also be used to lower the flushing overhead on a label change. With partitioned caches, each partition can have a register for its own security label. Then, the logic for a cache read only searches partitions with labels ℓ_P such that $\ell_P \sqsubseteq \ell_{cur}$. When the security label of a partition changes downwards in the lattice, only that partition will need to be invalidated.

HyperFlow has both *branch prediction*, which predicts whether or not branches are taken, as well as *branch target prediction*. The branch target predictor (BTB) in HyperFlow is fully-associative. The branch history table (BHT) has two-bit states per index and a global history register. Prior work has demonstrated that both forms of branch prediction create timing channels that are capable of leaking secrets from Intel SGX enclaves [**?**]. To prevent timing channels, when ℓ_{cur} moves downward, the BTB is invalidated and cleared, the BHT is cleared, and the global history register is reset.

D. Virtual memory

The HyperFlow implementation includes support for virtual memory. While HyperFlow protects memory using memory labels, the virtual memory system provides a familiar interface to the software and permits software to run on HyperFlow mostly unmodified. Virtual memory support includes instruction and data TLBs as well as a hardware page table walker (PTW). TLBs influence timing because they are caches of recently used Level-1 (L1) page table entries (PTEs). L1 PTEs store mappings from virtual to physical addresses. The PTW serves as a cache of L2 PTEs, which store pointers to L1 PTEs. The TLB and PTW state are labeled with ℓ_{cur}, and the state is cleared when ℓ_{cur} moves downward in the lattice.

Because the TLB and PTW state is labeled with ℓ_{cur}, PTEs must be stored in a memory page with a label that flows to ℓ_{cur}. This restriction must be satisfied by the software that manages the page tables. One simple option is to label the memory pages for page tables with $(C(\bot), I(\top))$, which is the least-restrictive information-flow label. The page table can also be stored in pages with more restrictive labels as long as they flow into the processes that use the page table.

E. Atomic memory operations

Like the baseline processor it extends, HyperFlow supports atomic memory operations (AMOs). AMOs are critical for operating system implementations because they are needed to implement synchronization primitives. AMOs are implemented by buffering both operands into different slots of the pending store buffer in the data cache before buffering the result back into the first slot of the pending store buffer. Implementing AMOs securely is challenging because the operands are buffered-in over multiple clock cycles, meaning they might have different timing labels, and because they are computed from operands that might also have different data labels. Because both AMO operands are buffered-in by the same instruction, both operands have the same timing label. To reduce the number of ports in the AMO execute unit, both operands, and therefore the output, are required to have the same label.

VI. Evaluation

We developed a prototype implementation of HyperFlow as an extension to a single-core configuration of the RISC-V Rocket Chip processor. This is a more full-featured processor than those previously implemented with statically checked information-flow [14], [20], [21], [43]. The prototype implementation label-checks with ChiselFlow and successfully runs all of Rocket Chip's ISA and application unit tests.

A. Processor features

The processor is pipelined, with branch prediction and branch target prediction. The branch history table has 2 bits of state per entry and a global history register. The branch target predictor is fully associative. The execution units include an ALU, a multi-cycle multiplier, and a floating-point unit (FPU).

The FPU is implemented as an independent coprocessor that receives instructions from the main processor, but it has its own independent instruction-decode unit, floating-point register file, and pipeline. The FPU sends requests to the memory hierarchy independently of the main processor.

The processor has four hierarchical protection rings, and a 32-bit virtual address space divided into 4KiB pages. The baseline processor has L1 instruction and data caches each with 64 sets and 4 ways. Both L1 caches have 2 pipeline stages. The data cache has a two-slot pending store buffer. Both caches are virtually indexed and physically tagged. The caches include cache controllers. Separate instruction and data TLBs store level-1 page table entries for each cache. A single hardware page-table walker refills both TLBs on misses and caches recently-used level-2 page-table entries.

Many of these micro-architectural features have been absent in prior information-flow secured processor implementations. To the best of our knowledge, HyperFlow is the first to include, TLBs, a PTW, branch and branch target prediction, and a pending store buffer. Most of these features introduce subtle timing channels that we address. HyperFlow is also the first to include data bypassing with fine-grained information flow labels. This necessitates dynamic label bypassing, which we must also label-check. The prototype implementation of HyperFlow includes all of the aforementioned features as well as the ISA and microarchitectural extensions described in Sections IV and V. The HyperFlow prototype does not include hardware accelerators and relies on a hard-wired memory controller on an FPGA.

B. Developer effort

HyperFlow was implemented by a single developer requiring roughly eight person-months of effort. While the existing Chisel implementation of the RISC-V Rocket chip provides most of the hardware functionality described above, it contains many timing channels. Much of the work required modifying the microarchitectural design to eliminate these timing channels, especially in the instruction and data cache pipelines, virtual memory hardware, and multiplier. Aside from, eliminating timing channels, many of the other hardware features, such as bypassing, required rewriting of the hardware to make it amenable to program analysis, even though the functionality was not changed. The labeling support provided by ChiselFlow was essential for removing timing channels correctly. By contrast, it was straightforward to extend the architecture with new instructions for managing labels, downgrading, and call gates – label-checking these features did not pose significant challenges beyond those of conventional hardware implementation. Label-checking many other features already present in RISC-V imposed no additional effort for label-checking; for example, the instruction expansion units and additional decode logic for compressed instructions label-check trivially. Label inference significantly reduced required developer effort.

C. Uses of downgrades

The RTL code for HyperFlow performs downgrades at various points; these downgrades are checked for nonmalleability by the ChiselFlow type system. Our formal results imply that insecure information flows can only arise because of these downgrades, which should therefore be inspected carefully. All but one downgrade is statically checked to be nonmalleable by the type system. Table III summarizes the uses of RTL-level downgrades. The first column shows the ISA-visible event to which the downgrade is tied, the second column states the number of downgraded expressions in the RTL code, and the third gives a brief description of what is downgraded. For all downgrades other than downgrades of data caused by explicit downgrade instructions, both an endorsement and a declassification happen.

We expand upon these descriptions here. When the processor resets (1), the register file tags are all initialized to $(\bot, \top)$. This initialization requires explicit writes to the tags because the register file labels are implemented as a sequential memory that can be synthesized as a BRAM on an FPGA. However, this initialization is secure because the processor is initially public and trusted and boots public and trusted code. For convenience, copies from the FPU to the integer register file (2) are automatically downgraded if the labels of the data coming out of the FPU can be nonmalleably downgraded. When ℓ_{cur} moves downwards in the lattice (3), it is possible for a single outstanding cache coherence transaction to remain in a pending transaction buffer, causing timing interference. We resolve this with a downgrade, but nothing is leaked if the software is written as described in Section IV: prior to lowering ℓ_{cur}, the software should issue a cache flush instruction to flush any buffered coherence transaction. When a memory transaction is issued (4), the data used to compute the address is downgraded to ℓ_{cur} because the address affects the timing of the cache transaction; this downgrade is for convenience because the address can otherwise be downgraded with an instruction. The label of the address is still protected by the data label, and so the store invariant in Table II is enforced by the data label. To permit the use of performance counters, writes to the CSR file (5) are downgraded to ℓ_{cur}.

The downgrading instructions (`DECLREG`, `ENDOREG`, `LWDWN`, `SWDWN` `DECLMEM`, and `ENDOMEM`) downgrade the stored data and the arguments to the instructions (6–9). These downgrades are done under a conditional statement that checks that these values are downgraded nonmalleably. As described in Section IV, the arguments are also downgraded to $(\bot, \top)$ because the arguments influence changes to public and trusted labels – this downgrade is also guarded by a nonmalleability check. The labels of the arguments are also bypassed, and bypassed labels are labeled ℓ_{cur}. Because the bypassed labels are inspected by the nonmalleability check, which influences whether or

	When is Information Downgraded	Number of Downgrades	What is Downgraded
1	On Reset	1	Register tags (for initialization)
2	FPU to Int instructions	1	Values copied from the FPU to integer registers (when nonmalleable)
3	ℓ_{cur} lowers	2	Presence of one outstanding finish coherence transaction
4	Memory instructions	1	Address is downgraded to ℓ_{cur}
5	CSR file writes	1	Data written to CSR file is downgraded to ℓ_{cur}
6	DECLREG, ENDOREG	7 (1 + 3 each)	Register contents, control signals, arguments
7	LWDWN	2	RF writeback data, dcache bypass data
8	SWDWN	1	P-store buffer data
9	DECLMEM, ENDOMEM	9 (1 + 4 each)	Page contents, control signals, arguments
10	RSTLREG	1	Control signal
11	REGLGATE	8	Control signals, arguments, pipelined data labels
12	LCALL, LCALLR	3	Control signal, arguments, pc value
13	LRET	1	Control signal
14	MMIO Responses	1	MMIO response transaction data

TABLE III
USES OF DOWNGRADES IN HYPERFLOW.

not the downgrade happens, the labels of the bypassed labels are also downgraded from ℓ_{cur} to $(\bot, \top)$. The control signals that induce the downgrades are also downgraded to $(\bot, \top)$ – this downgrade is always nonmalleable because these control signals are labeled ℓ_{cur}. The `LWDWN` instruction downgrades the data in two places in the core: the bypass data from the cache and the register file writeback data from the cache. The `SWDWN` instruction downgrades the stored data from the label in the pending store buffer to the label indicated by the memory tags that are stored in the cache. Neither `LWDWN` nor `SWDWN` changes the valuation of any labels, so these instructions do not induce downgrades of control signals or arguments.

Similarly, for instructions `RSTLREG`, `REGLGATE`, `LCALL`, `LCALLR`, and `LRET` (10–13), control signals are downgraded because these instructions affect public and trusted state. The `REGLGATE` instruction also includes a nonmalleability check on pipelined labels. The `LCALL` and `LCALLR` instructions store the old `pc` value in a public and trusted stack, so the `pc` is downgraded from ℓ_{cur} to $(\bot, \top)$.

Finally, one downgrade endorses and declassifies data from memory-mapped IO devices (14). This downgrade is not in general nonmalleable because we do not provide protection for or from memory-mapped IO devices.

D. Uses of dynamic checks

Dynamic checks are alternatives to downgrades. They are dynamic label comparisons that are written to establish some invariant. They are preferable to downgrades because they do not weaken security. However, care must be taken because dynamic checks should only be used when it is expected that the invariant can never be violated. Dynamic checks are used in HyperFlow to establish that $\ell_{lwr} \sqsubseteq \ell_{cur}$. This invariant is established in the control and status register (CSR) file where those registers are stored. However, the fact that this invariant is true is not visible in other components outside the CSR file. Another use of dynamic checks is to prevent timing channels caused by floating-point computation. Because the FPU computes on register values, and the time taken to finish a floating-point computation is data-dependent, the stall signals from the FPU are also data-dependent. The pipeline register stall signals in HyperFlow are labeled with ℓ_{cur}. We use a dynamic check that hides the stall signals from the FPU when the data values do not flow to ℓ_{cur}. Another dynamic check is used to convince the type system that the bypass value from the data cache does not cause timing channels; this dynamic check forces the bypass value from the cache to 0 if the timing label from the data cache response does not flow to ℓ_{cur}, but permits the actual data value to be returned otherwise. In practice, this dynamic check does not cause a functional error because when ℓ_{cur} is lowered, the data cache pipeline is stalled and cannot emit responses. Both the regular data cache bypass value and a downgraded bypass value produced by `LWDWN` are covered b the dynamic check.

E. RTL synthesis results

We synthesized the baseline processor and HyperFlow using Vivado v2016.2 targeting the 7z020clg484-1 FPGA found on the Zedboard Zynq 7000 development board. The baseline processor uses 34,508 LUTs (64.9%) on the FPGA, whereas HyperFlow uses 40,205 LUTs (75.6%), a LUT utilization overhead of 16.5%. The baseline processor uses 13 (9%) of the block RAM tiles whereas HyperFlow utilizes 19.5 (14%). The majority of the utilization overhead is due to the security tags stored with each cache entry, the tag table that associates tags with memory pages, and dynamic label comparisons, which are used for either access controls or dynamic checks. For both the baseline processor and HyperFlow, Vivado is able to meet a target clock frequency of 25MHz. For both designs, the critical path is through the FPU multiplier, so we expect that the minimum clock period is the same for both designs.

F. CPI Results

Although HyperFlow has no clock frequency overhead, there is performance overhead for timing channel protection. We measured the cycles per instruction (CPI) for HyperFlow when executing the RISC-V benchmark suite compared to the baseline RocketChip processor. For the HyperFlow processor, the processor executes with the same security level during the entire execution of the program. The results are summarized in Table IV. HyperFlow incurs a performance penalty because unlike RocketChip, the multiplier unit always executes in the worst case number of cycles. This performance penalty can be removed by disallowing multiplications of operands with data labels that do not flow to ℓ_{cur}. The mm benchmark

Benchmark name	HyperFlow CPI	RISC-V CPI	Percent Overhead
mm	1.089	1.063	2.4%
spmv	1.748	1.678	4.2%
median	1.631	1.284	27%
multiply	1.899	1.115	69%
qsort	1.542	1.531	0.7%
towers	1.052	1.030	2.14%
vvad	1.161	1.094	6.12%
dhrystone	1.206	1.187	1.6%

TABLE IV
PERFORMANCE RESULTS

is matrix-matrix multiply, spmv is double-precision sparse matrix vector multiply, median is a median filter, multiply does multiplications, qsort does quicksort on an array of integers, towers solves a towers of Hanoi puzzle, vvad adds two vectors, and dhrystone is the classic synthetic benchmark. The benchmark with the highest overhead is multiply, naturally. The geometric mean overhead is 12.4%.

HyperFlow also introduces performance overhead through hardware state that is flushed or invalidated on label changes, but these occur infrequently—when the active process changes via a context switch, or when the application information through instructions. The time between context switches is on the order of tens of milliseconds, so this overhead should be amortized over execution.

G. Usability

HyperFlow intends to support necessary communication among mutually distrusting principals in an environment managed by an operating system. HyperFlow also supports the expressive information flow label models that have been proposed for prior operating systems and languages for information flow control. In this section, we demonstrate that HyperFlow supports shared memory interprocess communication, communication through registers for system calls, and enforcement of rich information flow policies.

We demonstrate how labels represented in the FLAM model can be expressed as hypercube labels and enforced. The flow-limited authorization model (FLAM) is a recent model that supports decentralized security policies [2]. To illustrate the usability of the HyperFlow architecture, we implemented a simple application with a decentralized information flow control (DIFC) policy expressed in FLAM originally by Myers and Liskov [**?**]. DIFC policies allow communication among mutually-distrusting principals [5], [12], [26], [41].

The application emulates a tax-preparation service where a user ("Bob") sends data to a tax preparer and gets the result back. Both the tax preparer and the user are distrusting. Even though the tax-preparer process is allowed to perform computation on the user's data, HyperFlow prevents it from sharing the user's data or any values derived from it to any party other than the user. In our implementation, the tax-preparer process and the user process communicate through shared memory via IPC. Both processes are managed by trusted software implemented as a single system call that manages labels for the two parties. The application is implemented as assembly that runs in RTL simulations of our information-flow verified processor prototype. This result suggests that the HyperFlow ISA and prototype are sufficient to enforce complex application-defined information flow control policies with IPC and system calls.

Section VI-G1 provides background on FLAM. Section VI-G2 demonstrates how FLAM labels can be represented with the hypercube labels of HyperFlow. Section VI-G3 discusses how we use the IPC and system call primitives to construct the tax preparation application. Appendix B-1 discusses the system call and IPC primitives in more detail and shows code segments that illustrate how they work.

1) Background: The FLAM Label Model and Downgrading: FLAM unifies authorization and information flow polices. Principals p can delegate to each other; given principals p and q, if p *acts for* q, written $p \succeq q$, then p trusts q. Compound principals can be constructed from primitive principals. The conjunctive principal $p \wedge q$, denotes the combined authority of both p and q. Similarly, the disjunctive principal, $p \vee q$, represents the authority of either p or q. Principals together with $\succeq$ form a lattice, and $p \wedge q \succeq p \succeq p \vee q$ for any p and q.

In FLAM, principals are also information flow labels. The confidentiality of p is written $p^{\rightarrow}$, and intuitively represents the authority to observe secrets owned by p. The integrity of p, written $p^{\leftarrow}$, represents the authority to affect information owned by p. A second ordering on principals, defines permitted information flows. The statement p flows to q, written $p \sqsubseteq q$, denotes that information is permitted to flow from p to q. The ordering $\sqsubseteq$ forms another lattice over principals, which is orthogonal to the authority lattice. The meet and join in the information flow order are written $\sqcap$ and $\sqcup$. Any FLAM principal can be represented as a conjunction of a confidentiality projection and integrity projection $p^{\rightarrow} \wedge q^{\leftarrow}$. Labels of this form are said to be in normal form.

2) Mapping FLAM Labels to Hypercube Labels: FLAM labels are easily represented in the hypercube model using bit vectors. FLAM labels in normal form map directly to confidentiality and integrity components of hypercube labels. Primitive principals p are mapped to numeric constants, b_p. For example, if there are four 1-bit dimensions and p and q are mutually distrusting, one might map p to `1000` and q to `0100`. Figure 4 shows how compound principals can be mapped to hypercube labels. Here, $\mathcal{B}[\![p]\!]$ denotes the representation of p as a pair of its hypercube label components. The confidentiality component is the first in the pair, and the integrity component is the second. $\mathcal{B}_c[\![p]\!]$ denotes the confidentiality component of p and $\mathcal{B}_i[\![p]\!]$ is the integrity component. The values b_{min} and b_{max} are the lowest and greatest bit vectors that can be represented with the width of a label; they are respectively a sequence of all $1s$ and a sequence of all $0s$. Here, $\max_b$ is a function that computes the dimension-wise maximum of two hypercube labels, and $\min_b$ similarly computes a minimum.

3) Tax Preparation Application: To test the usability of HyperFlow, we implemented the tax-preparer application in assembly using the HyperFlow ISA. Bob has ℓ_{lwr} and ℓ_{upr} labels that are $B^{\leftarrow}$ and $B^{\rightarrow}$ respectively, and generally operates with a ℓ_{cur} label of B. The tax-preparer generally operates with ℓ_{cur} of $(B \wedge P)^{\rightarrow} \wedge P^{\leftarrow}$ because it is an instance of the tax

$$\mathcal{B}[\![p]\!] \triangleq (b_p, b_p)$$
$$\mathcal{B}[\![p^{\rightarrow}]\!] \triangleq (\mathcal{B}_c[\![p]\!], b_{max})$$
$$\mathcal{B}[\![p^{\leftarrow}]\!] \triangleq (b_{min}, \mathcal{B}_i[\![p]\!])$$
$$\mathcal{B}[\![p \wedge q]\!] \triangleq (\max_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \max_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\})$$
$$\mathcal{B}[\![p \vee q]\!] \triangleq (\min_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \min_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\})$$
$$\mathcal{B}[\![p \sqcup q]\!] \triangleq (\max_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \min_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\})$$
$$\mathcal{B}[\![p \sqcap q]\!] \triangleq (\min_b\{\mathcal{B}_c[\![p]\!], \mathcal{B}_c[\![q]\!]\}, \max_b\{\mathcal{B}_i[\![p]\!], \mathcal{B}_i[\![q]\!]\})$$

Fig. 4. Representing FLAM labels with hypercube labels.

preparation service specifically for handling Bob's requests, so it needs to be able to observe Bob's data. Its ℓ_{lwr} and ℓ_{upr} labels are $P^{\leftarrow}$ and $(B \wedge P)^{\rightarrow}$ respectively.

Before either Bob or the tax-preparer executes, a label manager that is fully trusted and public registers the `switch_process` call gate and initializes the memory label. Bob computes his tax form and sends the message to the preparer using shared memory as described in Section B-1. Bob then yields the processor by calling the `switch_process` gate so that the preparer can begin executing. The tax preparer receives the message and then computes the form using its proprietary data before declassifying the result. The preparer sends the result back to Bob via IPC and yields the processor back to Bob by calling the `switch_process` gate again. Finally, Bob receives the computed form.

VII. Related Work

Gate-Level Information Flow Tracking Gate-level information flow tracking [17], [27], [28], [33]–[35] applies information flow control to hardware designs at the gate-level. In the earliest variations of GLIFT [35], each gate of the hardware implementation is augmented with additional gates to track information flow. This approach incurs significant area and energy overhead. Later versions of GLIFT apply gate-level information flow tracking to simulated hardware designs [34], rather than to the implementation. This reduces overhead, but increases development effort compared to a conventional processor design flow. Because simulating every state in large designs is intractable, prior efforts to use simulation-based GLIFT check either small components [28], or limit the simulation to cover just the state space reachable with software that is co-designed with the hardware [33], [34].

Security-Typed Hardware Description Languages More recently, security-typed hardware description languages have been developed to check that information-flow policies are enforced at design-time. Unlike simulation-based approaches, type systems can ensure that the entire design is secure in just seconds. Sapper [20] and Caisson [21] are security-typed hardware description languages that generate GLIFT logic, but use a static analysis of the HDL code to minimize the amount of information flow tracking logic generated. Sapper and Caisson enforce security dynamically – security violations are converted into functional correctness violations. SecVerilog [13], [43] is a security-typed hardware description language that allows security policies to depend on run-time values, but enforces security statically by generating type errors. By enforcing security policies statically, hardware designers can avoid unexpected functional errors. The design of ChiselFlow closely follows the design of SecVerilog, and in particular, enforces security fully statically [13].

Information-Flow Secured Processors HyperFlow has strong information flow security guarantees because it is constructed with ChiselFlow, which has an information-flow control type system. Tiwari et al. [34] built the first processor with strong information flow security guarantees using a simulation-based approach to GLIFT. The processor supports just two security levels, and communication from the untrusted domain to the trusted one is not allowed. Similarly Zhang et al. construct a processor with two security domains [43]. Xun et al. [20], [21] construct a processor that supports a diamond lattice. None of these processors support communication that might violate information flow security. Ferraiuolo et al. [14] implement a processor that permits communication that weakens information flow security, but it does not constrain downgrades, and so security is weakened when downgrades are used. HyperFlow provides better assurance with downgrades because they enforce nonmalleable information flow control [4], a secure downgrading mechanism from prior work. None of these processors provide memory protection or privilege levels that can be arbitrary lattice-model information flow labels. Prior information-flow secured processors also do not have mechanisms for downgrading registers, or for control-flow transfers between different security domains; both of these mechanisms are necessary to support system calls.

VIII. Conclusion

This paper presents HyperFlow, a processor with strong information flow security that can be statically checked with an HDL-level security type system. Prior information-flow secured processors either enforce noninterference, which limits communication among distrusting processes, or provide no guarantee when communication would violate noninterference. HyperFlow provides communication that weakens noninterference, but constrains downgrades so that they are either robust or transparent. The HyperFlow architecture was also designed to limit downgrades so that they are mostly visible at the ISA level. HyperFlow was synthesized for an FPGA, and adds little area compared to a baseline processor. HyperFlow also enforces rich security policies expressible using a lattice; using them, we implemented an application with decentralized IFC policies.

References

[1] https://www.dropbox.com/s/u9r14lpwxyaudub/sirrtl-proofs.pdf?dl=0.

[2] O. Arden and A. C. Myers. A Calculus for Flow-Limited Authorization. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, June 2016.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, 2012.

[4] Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable Information Flow Control. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17.
[5] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.
[6] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, 2006.
[7] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, 2007.
[8] Intel Corporation. Intel Software Guard Extensions Programming Reference, 2014.
[9] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. ISCA '07, 2007.
[10] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, March 2008.
[11] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and André DeHon. Pump: A programmable unit for metadata processing. HASP '14, 2014.
[12] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. SOSP '05, 2005.
[13] Andrew Ferraiuolo, Weizhe Hua, Andrew C Myers, and G Edward Suh. Secure information flow verification with mutable dependent types. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 6. ACM, 2017.
[14] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17.
[15] Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. Wiley Publishing, 2015.
[16] J.A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE S&P*, 1982.
[17] Wei Hu, Dejun Mu, Jason Oberg, Baolei Mao, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Gate-level information flow tracking for security lattices. *DAES*, 2014.
[18] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. 1988.
[19] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.
[20] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In *ASPLOS*, 2014.
[21] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow. In *PLDI*, 2011.
[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.
[23] ARM Ltd. ARM Security Technology: Building a Secure System using TrustZone Technology.
[24] CVE-2017-5691, July 2017.
[25] Andrew C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *POPL*.
[26] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 2000.
[27] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Theoretical Analysis of Gate Level Information Flow Tracking. In *DAC*, 2010.
[28] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information Flow Isolation in I2C and USB. In *DAC*, 2011.
[29] Andrei Sabelfeld and Andrew C. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 2006.
[30] Andrei Sabelfeld and David Sands. Declassification: Dimensions and Principles. *JCS*, 2009.
[31] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Commun. ACM*, 1974.
[32] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, 2004.
[33] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution Leases: A Hardware-Supported Mechanism for Enforcing Strong Non-Interference. In *MICRO*, 2009.
[34] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a Usable Microkernel, Processor, and I/O System with Strict and Provable Information Flow Security. In *ISCA*, 2011.
[35] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete Information Flow Tracking from the Gates Up. In *ASPLOS*, 2009.
[36] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, 2002.
[37] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning, 2009.
[38] Rafal Wojtczuk and Joanna Rutkowska. Following the White Rabbit: Software Attacks Against Intel VT-d Technology, 2011.
[39] Steve Zdancewic and Andrew C. Myers. Robust declassification. CSFW '01, 2001.
[40] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.
[41] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.
[42] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.
[43] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *ASPLOS*, 2015.

APPENDIX A
CHISELFLOW IMPLEMENTATION AND EXTENSIONS

ChiselFlow extends Chisel, a domain-specific language for describing hardware designs embedded in the programming language Scala. Chisel is shallowly embedded in Scala. It is implemented as a set of Scala libraries that allow a user to design hardware. Chisel emits FIRRTL, an intermediate language that is compiled to Verilog, a widely used language for hardware design.

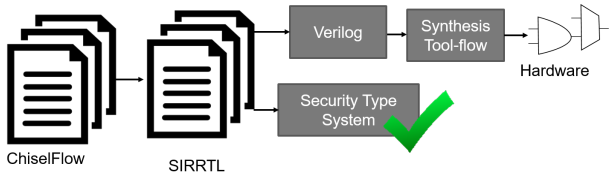

Fig. 5. ChiselFlow toolflow.

Figure 5 shows the toolflow for ChiselFlow. ChiselFlow extends the syntax of Chisel with features for describing and enforcing security policies. ChiselFlow emits SIRRTL, which extends FIRRTL with a syntax for information flow labels. Security enforcement is done by SIRRTL, which has an information flow type system. After type-checking, SIRRTL compiles to conventional Verilog. A conventional Verilog toolflow can then be used to generate an RTL simulation or synthesize to an FPGA or ASIC design.

Types that depend on the run-time values of signals are statically checked by using a program analysis that relies on the SMT solver Z3 [10] for dispatching proof obligations. The program analysis models the behavior of the hardware signals that are propagated in parallel. The program analysis generates a set of Z3 constraints that model the value assigned to each signal—for each signal, a single expression is generated by unrolling conditional statements. Cases in which sequential variables retain their value from the previous cycle are modeled by an auxiliary variable that represents the old value from the previous cycle. In an accompanying technical report, we formalize a core subset of SIRRTL and prove that well-typed hardware modules do not leak secrets aside from downgrades [1].

A. Heterogeneously Labeled Data Structures

Hardware modules written in Chisel commonly group signals together in bundles, which are analogous to structs or record types. ChiselFlow supports heterogeneously labeled bundles. The syntax of ChiselFlow allows each signal within a bundle to take an additional argument that describes the label of that individual field. Bundle labels in ChiselFlow are similar to the path labels in Jif [7]. Because the program analysis used in ChiselFlow dispatches proof obligations to Z3, bundle labels must be represented in Z3. We represent bundles using the algebraic datatype theory.

Chisel also includes the `Vec` type for describing arrays. In addition, Chisel has a `Mem` type for describing memories, which can either be used to instantiate BRAMs on an FPGA, SRAMs in an ASIC design, or arrays of registers. To support heterogeneously labeled `Vecs` and `Mems` in ChiselFlow, we adopt a recent proposal in which arrays in a secure hardware language can be labeled with functions that map the index of the array to the label of the element at that index [14].

B. Nonmalleable downgrades

Much like the downgrading instructions in HyperFlow, ChiselFlow supports robust declassification and transparent endorsement with syntax `decl`(e, ℓ) and `endo`(e, ℓ) respectively in which e is an expression and ℓ is a label. The typing judgments for these expressions closely resemble those used in a recent functional programming language for information flow control called NMIFC [4]. A difference between the typing judgments of downgrades in ChiselFlow and NMIFC is that ChiselFlow must use the program analysis to reason about the run-time valuation of signals mentioned in the labels.

```
# Set Page Labels. cur_lvl: {\bot-> & \top <-}
li x1, 0x84       # {B-> & P<-}
li x2, 0x48       # {P-> & B<-}
la x3, prep_to_bob
la x4, bob_to_prep
setmem x1, 0(x3)
setmem x2, 0(x4)

...

# Bob Sends. cur_lvl: {B}
la x5, bob_to_prep
swdwn x6, 0(x5) #decl {B} to {P-> & B<-}

...

# TP Receives. cur_lvl: {(B&P)-> & P<-}
la x5, bob_to_prep
lwdwn x6, 0(x5) #endo {P-> & B<-} to {(B&P)-> & P<-}
```

Fig. 6. IPC Example.

C. Label inference

The baseline processor that HyperFlow extends is many lines of code. To aid in the effort needed to label HyperFlow, ChiselFlow supports label inference. In ChiselFlow, only the module ports need to be explicitly annotated with labels whereas the labels of internal registers and wires are inferred. ChiselFlow is the first security-typed hardware language with support for label inference, though the label inference algorithm is similar to those of prior security typed languages for software [25]. Initially, all internal signals that are not explicitly labeled are given variable labels $v \in \mathit{VarLabel}$ that represent unknown labels. Initially, variable labels have the least restrictive label $\top$. Label inference then proceeds as a work-list algorithm that iteratively lowers the estimate of the final label of each variable label based on the labels of other signals it influences either directly or through implicit flows.

Appendix B
IPC and System Calls

1) Interprocess Communication: Figure 6 shows an example of how messages are communicated among processes in the tax-preparer application, and more generally, shows how shared memory IPC works in HyperFlow. In the example, $0b0100$ represents the principal Bob (B), and $0b0010$ represents the Tax Preparer principal (P). A page of memory is allocated for Bob to send messages to Preparer with label $B^{\rightarrow} \wedge P^{\leftarrow}$, and for Preparer to send messages to Bob with label $P^{\rightarrow} \wedge B^{\leftarrow}$. Public and trusted code initializes the labels of the pages used for IPC. In the code segment shown, Bob has a ℓ_{cur} label of B. Because the Tax Preparer is an instance of the tax preparation service specifically for handling Bob's requests, it has a ℓ_{cur} label of $(B \wedge P)^{\rightarrow} \wedge P^{\leftarrow}$ so that it can see Bob's data. For Bob to send a message to Preparer, it simply performs a `SWDWN` instruction on a register with label B which downgrades the register contents to the label of the destination page ($P^{\rightarrow} \wedge B^{\leftarrow}$). This downgrade is robust because Bob has sufficient integrity to remove the $B^{\rightarrow}$ component of the label. The $P^{\rightarrow}$ component of the label can be added because this increases the restrictiveness of the label. The Preparer receives

```
# Gate Registration: cur_lvl {\bot-> & \top<-}
la x1,      switch_process
li x2,      0x0F # {\bot-> & \top<-}
reglcall x1, x2

# Bob. cur_lvl: {B-> & B<-}
...          # compute form, store in shared page
li x4, 0x0
la x3, swith_process
declreg x1, x4 # Flag to choose Bob or Preparer
declreg x2, x4 # Address to jump to after call
lcallr 0(x3)

# Process Switch Call: cur_lvl {\bot-> & \top<-}
switch_process:
li x4, 0xF
endoreg x1, x4 # Flag to choose Bob or Preparer
endoreg x2, x4 # Address to jump to after call
...           # Set labels, jump to target
```

Fig. 7. Syscall Example.

the message by doing a LWDWN instruction which endorses the integrity of the message to $P^{\leftarrow}$.

In some cases, it is possible to receive a message through IPC by first downgrading a register and then doing a conventional load instruction to the downgraded register. However, this example demonstrates that this is not always possible, and that the LWDWN instruction is necessary for expressiveness of the ISA. The tax preparer cannot endorse the integrity of a register to $B^{\leftarrow}$ because its label does not flow to $B^{\leftarrow}$. However, it can endorse the $P^{\rightarrow} \wedge B^{\leftarrow}$ data to $P^{\leftarrow}$, so it can receive the data with a LWDWN instruction.

In this example, we used two separate pages for communication in each direction. However, it is more conventional for processes to share a single page that both processes can both read and write. The label model of HyperFlow is also expressive enough to support bi-directionally shared pages—a single page could be labeled $B \vee P$. With this label, both B and P can write to the page, and both process can read from the page by endorsing it. However, with the aforementioned numerical representations of B and P, $B \vee P$ computes to the fully public and fully distrusted label, which any other process can read and write as well. By representing B as $0b01001$ and P as $0b00101$, $B \vee P$ becomes the more restrictive label, $0b00001$. Because this has a one-bit overhead in the size of labels, we chose to use the more compact label encoding and use two pages for communication. Other representations are also possible. For example, using 2-dimensional, rather than 4-dimensional hypercubes offers a compact encoding while retaining unique disjunctions, however, it reduces the total number of physical principals.

2) System Calls: In the tax preparer, a single trusted system call is used to manage Bob and the Preparer by initializing labels for the target process before jumping to the entrypoint of the process. This system call is implemented as a call gate. Figure 7 shows a small segment of the call gate as well as how the gate is registered and called, and by extension, the example shows how system calls can be implemented in HyperFlow more generally. Initially, fully trusted and fully public code registers the call gate at address switch_process and with label $\bot^{\rightarrow} \wedge \top^{\leftarrow}$. The call gate takes two arguments. One describes whether labels should be initialized for Bob or the Preparer, and the other is the PC value that is the entrypoint for the next process. When Bob is done computing, it executes the call gate. Before doing so, it must declassify the confidentiality of the two arguments from $B^{\rightarrow}$ to $\bot^{\rightarrow}$. It then simply calls the gate with an LCALLR instruction.

At the start of the call gate, the handler must endorse the integrity of the two arguments from $B^{\leftarrow}$ to $\top^{\leftarrow}$. The call gate handler then sets the tags of all registers and the levels of ℓ_{nlwr}, ℓ_{ncur}, and ℓ_{nupr} to values that depend on the next principal to execute. It then does a SETBOUNDS instruction before jumping to the entrypoint of the next process.

In conventional processors, system calls work by first jumping to trusted code that contains a system call handler table – the particular call handler to execute is selected by using a register argument that contains the call number. This model is also supported by HyperFlow. However, because HyperFlow replaces conventional privilige modes with lattice model information flow labels, the call gates of HyperFlow are more general and can both improve performance and the precision with which access controls are enforced.

APPENDIX C
DISCUSSION

In this section we discuss some of the design and implementation trade-offs we made in order to satisfy the goal of making HyperFlow label-check with ChiselFlow. This section also discusses the process of implementing a processor that label-checks and how this process differs from conventional hardware design without label-checking.

Labels of Labels In ChiselFlow, wires can be used to represent information flow labels that can change at run-time. Because these wires are still wires, they are also labeled. In most information flow systems, it is assumed that information flow labels are fully trusted and fully public. If the labels cannot be trusted, then they clearly cannot be used to establish security, and if they contain secrets, even inspecting them to implement access controls might leak those secrets.

However, because HyperFlow is timing-channel free and implements fine-grained per-register and per-page data labels, we found it necessary to give more restrictive labels to some of the signals that represent data labels. For example, the per-page data labels are stored in the cache and used as security types for the data in the cache. Most control signals in the cache are labeled with ℓ_{cur} because they represent signals that affect timing. Because the values of these control signals influence the time that per-page data labels are brought into the cache, the labels themselves must be labeled with ℓ_{cur}.

Register File Labeling The register file tags underwent several revisions even though it is a simple component. This is related to the issue of labeling labels because the register file tags are the labels of the registers. Initially, we labeled the register file tags with ℓ_{cur} following the same design choice that we made for the data labels in the cache. However, this would have required the register file tags to

be cleared whenever ℓ_{cur} moves downward in the lattice, and by extension the associated registers would need to be cleared. Clearing the registers whenever ℓ_{cur} was lowered would cause unacceptable limitations in the expressiveness of the ISA.

We later tried labeling the registers with ℓ_{lwr}. With this labeling, the level of ℓ_{cur} can move freely without clearing the tags because $\ell_{cur} \sqsubseteq \ell_{lwr}$ at all times. The tags (and registers) need only be cleared when ℓ_{lwr} is changed—effectively, whenever the active process is changed. However, the tags could only be set when $\ell_{cur} = \ell_{lwr}$. This design point is plausibly acceptable, but it still imposed too many restrictions. For example, it is useful to set register tags while at a level above ℓ_{lwr} so that a spare register does not have to retain the value of ℓ_{lwr}. It is also potentially useful to avoid clearing the tags even when ℓ_{lwr} changes.

In our final design, the register file tags (though not the general purpose registers themselves) are labeled totally public and totally trusted. As we discuss in VI-C, the control signals from the instruction decode unit that determine when tag-setting instructions happen are downgraded. Given that these downgrades only happen during particular instructions and that the information released is clear, these downgrades do not violate our design goal of making information release ISA-visible.

Automatic Tag Propagation Initially, we expected that security labels could be propagated automatically. For example, following an `ADD RS3, RS2, RS1` instruction, we would like to compute the join of the labels of `RS1` and `RS2` and store the result in `RS3` without needing explicit instructions to set the tag of `RS3`. In fact, this form of dynamic tag propagation is common in tagged hardware architectures [9], [32]. However, whether or not general purpose registers are updated depends on control signals that are influenced by timing and labeled with ℓ_{cur}, but the labels are public and trusted. In other words, dynamically updating the security tags themselves introduces subtle timing channels. We found it preferable to only allow register tags to be updated by explicit instructions that are controlled by the software.

Multi-Cycle Execute Unit Stall Signals HyperFlow has two execute units that take multiple cycles to perform a computation, the multiplier and the FPU. The time to complete these computations depends on the data values. Because the data values have security labels that might not flow to ℓ_{cur}, the time to finish these computations could create timing channel vulnerabilities if they are not carefully controlled. In the HDL code, this timing channel is visible as a flow from the operands from the register file (which have labels that depend on their tags) to the stall signal, which has the label ℓ_{cur}. We address this timing channel for each of the two execute units in different ways. For the FPU, we do not permit computations on operands with labels that do not flow to ℓ_{cur}. For the multiplier, we permit computations on operands that do not flow to ℓ_{cur}, but the operations always complete in the worst-case time. This presents a tradeoff between the expressiveness of the ISA and performance. We chose to take two different approaches for each primarily to demonstrate that either can be statically checked with the information flow type system.

For the FPU, when the labels of the operands do not flow to ℓ_{cur}, the stall signals in the pipeline are modified to hide the stall signal that is an output from the FPU – this dynamic check converts the insecure information flow to a correctness error if the arguments to the FPU ever have operands with labels that do not flow to ℓ_{cur}. Access controls guard the inputs to the FPU to ensure that this invariant always holds, and a correctness violation is never introduced in practice.

The multiplier is modified so that it always terminates in the worst-case time. In this way it is always correct, but does not leak information through timing. Unlike the FPU, multiplications can be performed on operands with labels that do not flow to ℓ_{cur}. However, implementing a constant-time execute unit is not straightforward. Because the multiplier is a simple FSM it is a better candidate for this approach than the FPU, which can be viewed as an independent coprocessor. To implement a constant-time multiplier, we used two state registers: the primary state register and the shadow state register. The primary state register always points to the FSM state farthest from the terminal state. The shadow state is always updated based on the control signals and operands to point to what the state *would have been* in the original multiplier. The primary state does not depend on the operands, and so its label only depends on the label ℓ_{cur}. The label of the shadow state does depend on the label of the operands, and the value of the shadow state is used to compute the output, but the stall signal does not depend on the shadow state. In this way the multiplier label-checks and is correct.

HyperFlow and the Spectre and Meltdown Attacks The recent Spectre [19] and Meltdown [22] attacks exploit speculate execution. HyperFlow, like the RocketChip baseline it extends, does not support out-of-order execution or speculative execution. It does, however, speculatively update the branch history table (BHT). The BHT in RocketChip includes a table of 2-bit counters per index and a global history table. On a hit in the branch-target buffer (BTB), the BHT is updated speculatively. Although the BHT and BTB are accessed in the fetch stage of the processor pipeline, the speculative update is not undone until the memory stage by reseting the global history table on branch mispredictions. In an unsafe design, it might be possible for a secret instruction to cause a speculative update to the BHT that is visible to a public instruction earlier in the pipeline, leaking information through timing. In HyperFlow, however, when ℓ_{cur} moves downward in the lattice, the BTB is invalidated and cleared, and both the BHT global history register and branch table are cleared in the same cycle that ℓ_{cur} changes.

The Meltdown attack exploits a vulnerability in memory permissions checks in the data caches of Intel processors [22]. In some Intel processors, the data fetch and TLB permissions checks that a memory access entails are implemented with separate micro-ops. Because the data access can happen before the permissions check, it is possible that the data access can modify the cache even though the TLB permissions check later rejects the access. The speculatively modified cache

state creates a timing channel. In HyperFlow, a potential similar vulnerability is prevented in two ways. First, the cache implementation is guarded by a timing label that represents the secrecy of the process that brought the entries into the cache. When ℓ_{cur} becomes lower than HyperFlow, the cache is invalidated. Second, permissions checks that govern memory data are tightly coupled with data access. The memory tags governing the accessed data are inspected in the same cycle of the cache pipeline during which the data is granted.